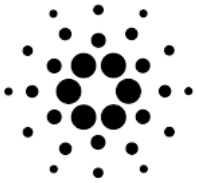

Unified Backend Architecture

in Cardano SL

Abstract

This article provides technical specification for Unified Backend Architecture in Cardano SL.

Contents

Author and Document Owner 4
Status 4
Proposal 4
Motivation 4
Prerequisites 4
Assumptions 4
Requirements 5
Key Idea 5
Multi-Tenant Architecture 5
Scalability 5
User Stories 6
    Just after Installation 6
    «I want new wallet to use ADA» 6
    «I want to have an access to my money from another device» 6
    «I lost my hardware wallet and want to restore an access to my money» 6
    «I want to see how much money I have» 7
    «I want to receive money from other user» 7
    «I want to send money to other user» 7
API Extension 7
    Create Unsigned Transaction 7
    Send Signed Transaction 7
Wallet State 8
Authentication Scheme 8
    HTTPS 8
    Session 8
Wallet Recovery 8
    Trees of Depth 1 9
    Taller Trees 10
Crypto Operations 10

Security and Possible Attacks 10
MITM 10
Other Attacks 10

Author and Document Owner

Denis Shevchenko

Status

DRAFT

Proposal

We propose to create a lightweight Daedalus client. And it is possible to unify the different use cases under a single architecture, starting from the crucial observation that both interfacing with an hardware wallet or provide a mobile client can be seen as a specialisation of the general case of having a “keyless” wallet backend operating with the private keys and cryptography “hosted” on the client side. It will allow to keep blockchain on a backend only.

Motivation

Currently Daedalus can be treated as a heavyweight client: it works in a pair with Cardano node which stores full copy of the blockchain. It leads three problems:

1. Such a heavyweight client must have enough disk space to store growing blockchain.
2. Cardano node requires quite a lot system resources.
3. During (first) start heavyweight client must sync (new) blocks and this is long process, even with good internet connection.

These problems make impossible to install such a heavyweight client on a mobile device. The purpose of Unified Backend Architecture is to solve these problems, while preserving a common architecture with the heavyweight desktop client and heavyweight desktop client with hardware devices.

Prerequisites

The reader should be familiar with following concepts:

1. Daedalus frontend,
2. wallet backend web API,
3. work with wallets,
4. transaction processing.

Also it is assumed that the reader knows basic terminology from Cardano Docs, for example, UTXO.

Assumptions

N/A, see **Prerequisites**.

Requirements

Unification of the wallet backend architecture should allow to:

1. work with different hardware wallets in the same manner,
2. create lightweight Daedalus mobile clients (iOS/Android),
3. create lightweight Daedalus desktop clients (Windows/macOS/Linux).

The main benefit of such a lightweight client is independence from the growing blockchain: the client does not store blockchain locally and does not wait for a "Syncing blocks".

Moreover, the unified architecture can also cover the existing desktop heavyweight solution, running the wallet backend on the same machine. This is what will give us maximum sharing within the codebase.

Key Idea

As mentioned before, the key idea is to unify the different use cases under a single architecture, we mean hardware wallets, mobile clients, lightweight desktop clients and heavyweight desktop clients.

In the case of Ledger (or any other hardware wallet) the privte keys and the cryptography are hosted on the device, whereas in a mobile client they are held on a smartphone. And this schema fits for "featherweight Daedalus desktops" that do not want to host the growing blockchain on the computer's storage.

More specifically, a Daedalus mobile client will consist of an iOS or Android application which will communicate with a cluster of nodes that will keep the global UTXO state for the blockchain and that will be capable to reconstruct each mobile wallet state without hosting any user-specific data.

Multi-Tenant Architecture

Mobile clients will communicate with a clusters of nodes, so the basic requirement is an ability of one node to work with particular number of clients simultaneously. For example, if 1 cluster has to serve 10000 mobile clients and this cluster consists of 100 nodes, each node has to be able to work with 100 clients simultaneously and reliably. Thus, the longest procedure is a new payment (creating new transaction), and even if all these 100 clients will sign their transactions and send it for publishing at the same time, the node must be able to handle these transactions reliably and as fast as possible.

To achieve this goal the node must implement robust concurrent model. More specifically, all requests to the wallet web API endpoints must be handled concurrently. In addition, it is possible to use queues/buffers, it will allow to keep sustained transactions submission rate.

In case of correct design decisions, even the slowest endpoints, like `/api/v1/transactions`, can be handled fully concurrently (publishing to the network, storing in the `wallet-db`, etc).

Scalability

Initially clusters of nodes will be hosted by IOHK (on AWS EC2). And we need a mechanism of auto-scaling of the cluster: it must be able to grow proportionally to growing number of clients.

Probably we need an auto-scale supervisor. Suppose we have 1 cluster that consists of 100 nodes and each of node can reliably serve 100 clients. If supervisor sees that number of clients connected to this cluster is growing (and becomes more than 10000), it launches additional nodes. But exact

resource and tuning to do to be decided during later benchmarking. However, we need benchmarks to find bottlenecks in autoscaling.

Launching additional node is relatively expensive action (in terms of time and memory). So it is better if one single node will be able to serve more clients than just increase a number of nodes in the cluster. We have to understand what the real bottlnecks are. E.g. for relays, the number of clients syncing concurrently *is* the bottleneck, not the number of client just keeping up with the blockchain.

User Stories

These are typical user stories (workflows), with description of API calls.

Just after Installation

After installation of mobile client application user does not see any wallets, and he must create at least one to work with a client application.

During the first connection with the node the new session is establishing, please read an explanation below.

«I want new wallet to use ADA»

First of all, user has to create a new wallet. It produces a POST-request to `/api/v1/external-wallets` (with information about client's public key), node creates a new wallet in `wallet-db` and sends response with information about new wallet.

Even if node N which serves the client C will die, another node M (the client C will reconnect to) will be able to handle information about C's wallet(s), because `wallet-db` is storing in the cluster.

«I want to have an access to my money from another device»

Wallet can be imported using client's secret key.

Since secret keys remain only on the client, import is a frontend-only operation: just turning the multi-word passphrase (or backup phrase) into the root keypair.

«I lost my hardware wallet and want to restore an access to my money»

If user changed a smartphone or lost hardware wallet, it is possible to install mobile Daedalus client (or buy new hardware wallet device) and restore user's wallet using backup phrase (for example, 12 words) generated during this wallet creation.

Wallet restoring produces a POST-request to `/api/v1/external-wallets`, and node checks if such a wallet exists. If so, response contains an information about the wallet.

For more details about wallet restoring process please see **Wallet Recovery** section.

«I want to see how much money I have»

User can see its current balance (by default, the balance should always be shown in the client application).

After client application launching and after transactions client is asking for current balance. So we need an information about current wallet, because balance is a part of such an information.

GET-request to `/api/v1/wallets/WALLET_ID` returns an information about particular wallet.

Please note that user can have few wallets with different balances, so balance for current active wallet will be asked and shown.

«I want to receive money from other user»

User can receive a payment from other user. To do it he must provide a receive address.

POST-request to `/api/v1/external-addresses` returns new receive address which can be used to receive a payment from other user.

«I want to send money to other user»

There are 3 steps to make a new payment:

1. create transaction,
2. sign transaction,
3. publish transaction.

When user creates a new payment (by defining recipient's address, amount and fee), it produces a POST-request to `/api/v1/external-transactions/unsigned`, and response will contain a new transaction without a witness (signature).

After that user signs new transaction using its private keys (hosted on the client side).

Then user send a payment. It produces a POST-request to `/api/v1/external-transactions`, node receives transaction with a signature, forms transaction witness and publish this transaction to the Cardano network as usually.

API Extension

API v1 must be extended to support both the mobile clients and the hardware wallets.

Create Unsigned Transaction

When user wants to send money to other user, he creates a new payment. It produces a POST-request to `/api/v1/external-transactions/unsigned` endpoint, and response will contain a new transaction without a witness (signature). In addition, response will contain the HD (Hierarchical Deterministic) wallet indexes of the keys, this information is needed to sign new transaction.

Send Signed Transaction

After user signs new transaction using its private keys (hosted on the client side) he sends it with a signature. It produces a POST-request to `/api/v1/external-transactions`, node receives transaction with a signature, forms transaction witness and publish this transaction as usually.

Wallet State

An analysis of the computational complexity of rebuilding the state of a wallet (e.g. its balance and payments history) on the server in a way as much stateless as possible.

Authentication Scheme

HTTPS

Client must be sure that server is not a malicious one, so we use HTTPS-connection with a standard check of TLS certs on the client side.

Session

After successful TLS checking client and server establish a new session.

There will be a server-side encrypted token generated when a session is established between a client and a generic server. To generate this token the user will feed to the server some form of secret, probably a user generated password or an app/system generated password.

This token will allow a single client to handle and perform operations on multiple wallets at the same time.

This token will be held on the server side and probably used as a key for a `ClientStateMap` which will allow to cache client-specific info, including a list of addresses known to the server and that belongs to a given client.

Sessions can expire and the entry is removed from the `ClientStateMap`, which requires the user session to be recreated again.

Sticky sessions either at the load balancer' level or at the RESTful HTTP location' level to ensure that a client talks as much as possible with the same server.

There will be a Proof-Of-Ownership between the client and the server. This is needed to ensure that a (malicious) client will not "peek" into a portion of the server's state it does not control. One practical example would be a client asking about a balance of an address he does not own.

Some messaging will be done to accommodate the fact standard HD-wallet derivation uses elliptic curves public/private addresses, but we are dealing with Cardano addresses and we need to take into account things like delegation information, but these should not affect how authentication is carried out.

Wallet Recovery

As was mentioned earlier, wallet can be restored. In order to reconstruct a wallet, all addresses belonging to that wallet which appear on the blockchain need to be identified. In the current implementation, this is done by traversing the blockchain, and for each address, checking whether it belongs to the wallet. Unfortunately, this is linear in the size of the blockchain, leading to a very poor user experience.

To speed this up, we will reverse the strategy. Instead of going through the addresses on the blockchain, checking for each whether it belongs to the wallet, we go through the addresses in the wallet, and search whether they appeared on the blockchain.

In order for this to be efficient, we need to maintain an index, where we can look up addresses in the blockchain by some key, and we need to have a way of generating the key for an arbitrary range of addresses in the wallet, using only the root key as input.

Of course, we cannot search for all possible addresses of the wallet. Instead, we utilise the tree structure of the HD wallet. We will require that the wallet software populates this tree in a specified way that will allow us to do a kind of exponential search for the addresses of the wallet.

In Bitcoin and other proof of work systems, it is possible to just generate a range of public keys, and those translate directly into addresses. We cannot do that here, since the addresses will also carry delegation information, which is additional information not derivable from the root key.

Instead, we choose the HD wallet tag ht as the key for address lookup. ht is a string contained in the address, the form of which can be chosen by the wallet implementation. We use the scheme:

ht = H(root private key||HD path)

where H is a hash function, and HD path specifies the location of the address within the HD wallet tree.

Trees of Depth 1

To simplify, let us consider a wallet where the HD wallet tree is of depth 1, so that each address has an index $i \in \mathbb{N}$. We will require that the wallet creates addresses in order, and that there is a maximal address gap $\bar{i}$, such that the address α_i will not be generated unless there is an address $\alpha_{\hat{i}}$, with $\exists \hat{i} \in [i-\bar{i}-1, i-1]$ already appearing on the blockchain.

The first step in restoring a wallet is to find an upper bound on the number of addresses of the wallet, i_{up}. This can be done by consecutively looking at the intervals

$I_n = [2^n + i | i \in [0, \bar{i}]], n \in \mathbb{N}$

and checking whether any of the addresses in α_i for $i \in I_n$ appears on the blockchain. This check is performed by creating the corresponding HD wallet tags, and doing a look-up in the index. For some n, this will fail, and we will have found $\bar{i}$ consecutive indices for which there are no addresses of this wallet on the blockchain. Because $\bar{i}$ is the maximal address gap, no address larger than 2^n has been created for the address, and we have $i_{up} = 2^n$.

Afterwards, we can perform a binary search for the maximal address i_{max}, in the interval $[2^{n-1}, 2^n]$. In each step of this binary search, we will probe for $\bar{i}$ consecutive addresses, starting from an offset i. If none of them exist, we know that $i_{max} < i$, otherwise $i_{max} \geq i$. Finally, we will create all tags ht in the range $[0, i_{max}]$, and look up the corresponding addresses.

Two remarks are in order:

1. **Early Finish and Memoisation** The above process will perform more lookups than necessary. The binary search can be aborted once the search window gets smaller than $\bar{i}$. In addition, we should consider memoising the lookups.
2. **False Positives** We perform lookups by ht, which is a hash of the private root key and HD path. The output size of the hash function is kept small in order to have short addresses, but that means that collisions will likely happen. A hash collision will lead to a false positive, wrongly identifying an address as belonging to the wallet, so we have to include a check.

   For each address we are looking up, we know the HD path, and we can read the staking information from the address itself, so this allows us to create the whole address and detect a false positive.

   Since false positives can only increase, not decrease, the range of addresses that we are going to create, it is sufficient to perform this check in the last step, when lookin up all addresses in $[0, i_{max}]$.

Taller Trees

This scheme can be generalised for trees of larger depth. The current wallet in Cardano has a fixed depth of 2. Each address in this wallet has an $index(i, j) \in \mathbb{N} \times \mathbb{N}$. In order to generalise the above wallet restoration procedure for this wallet, we will require that there is no gap in the i, and a maximal gap $\bar{j}$ in j.

Identifying the maximal value i_{max} is straightforward: look at lists of indices

$[(i, j)|j \in I_0]$

for increasing values of i, until there is no address found on the chain for a specific value of i. Once i_{max} is found, we can iterate the method for trees of depth 1 over all $i \in [0, i_{max}]$.

Further generalisations to arbitrary depths are straightforward, provided that

- all the leaves are at the same depth,
- at each depth, we can require a certain maximal gap.

Crypto Operations

A description of how the main operations on a crypto-wallet will be implemented, according to the final index derivation mechanism chosen (sequential vs random).

We need to use deterministic (sequential or deterministically seeded PRNG).

Security and Possible Attacks

MITM

To mitigate the impact of MITM attacks we have to include a fee in transaction explicitly. This will also helps the client in case of malicious/compromised wallet backend.

Currently it is impossible, because transaction contains just inputs, outputs and attributes, and we have to know current UTXO to calculate a fee. So we must change transaction's format for explicit including a fee in it.

Other Attacks

This should eventually be reviewed by the security internal auditors.